Análisis de Eficiencia - Tiempo de Ejecución

Pablo Castro
Estructura de Datos y Algoritmos
UNRC

Propiedades Deseables

Hay ciertas características deseables de los programas:

- Corrección — Que el programa haga lo que dice su especificación
- Simplicidad — Que sea fácil de comprender
- Eficiencia en tiempo de ejecución — Que dada cualquier entrada, el tiempo de respuesta del algoritmo no sea excesivamente largo
- Eficiencia en recursos — Que no ocupe demasiada memoria
- Adaptabilidad — Que sea fácil de extender o modificar
- Facilidad de uso — Que sea fácil de usar

Importancia de la Eficiencia

Aunque los procesadores son cada vez más rápidos, la eficiencia es importante por:

- Existen soluciones a problemas simples que son extremadamente ineficientes (para cualquier procesador).

Por ejemplo Fibonacci

- Los algoritmos eficientes nos permiten resolver más problemas en menos tiempo.

El avance en hardware es solo polinomial

- La tesis extendida de Church-Turing.

Un Ejemplo

Supongamos que queremos calcular 2^n :

```
/**
* Version ineficiente de f(n)=2^n
*/
public static long exp1(long n){
    if (n == 0)
        return 1;
    else
        return exp1(n-1) + exp1(n-1);
}
```

Para valores grandes de n se vuelve imposible de ejecutar

Este programa es correcto pero ineficiente!

Cómo medir el tiempo...

Podemos pensar en varias alternativas:

Depende de la plataforma y la entrada

- Correr el programa y contar la cantidad de tiempo que tarda.

Depende de la entrada

- Calcular la cantidad de operaciones elementales que realiza el prog.

Puede no reflejar el tiempo de ejecución real

- Calcular la cantidad de operaciones elementales en función del tamaño en el peor caso

Difícil para calcular

- Hacer lo mismo pero en el caso promedio

Análisis en el peor caso

Cuando hacemos análisis en el peor caso debemos:

- Determinar que es la longitud de las entradas del programas.

  Bastante directo: longitud de la lista, altura de un árbol, etc

- Determinar el peor caso del programa.

  Hay que analizar el código

- Determinar cuales son las operaciones básicas del programa

  Directo: asignaciones, operaciones aritméticas, etc

  Depende del nivel de abstracción que trabajemos

Función de Crecimiento

Dado un programa podemos definir su función de crecimiento:

$$T_p : \mathbb{N} \to \mathbb{N}$$

Función de crecimiento del programa P con una entrada de tamaño n

- Esta función refleja el tiempo de ejecución del programa con respecto a el tamaño de entrada
- Su definición depende del programa a analizar
- Suponemos que las funciones son monótonas:

$$n \leq m \Rightarrow T_p(n) \leq T_p(m)$$

Ejemplo

Consideremos el SelectionSort:

```
int i = 0;
while (i < MAX_ARRAY) {
    int max, indice_max, j, aux;
    max = A[i];
    indice_max = i;
    j = i;
    while (j < MAX_ARRAY) {
        if (A[j] > max) {
            max = A[j];
            indice_max = j;
        }
        j++;
    }
    aux = A[i];
    A[i] = A[indice_max];
    A[indice_max] = aux;
    i++;
}
```

Ordena un arreglo de valores comparables

Análisis del Selection

- Tamaño de entrada: La longitud del arreglo a ordenar.
- Peor caso: El arreglo está ordenado en forma creciente.
- Operaciones Elementales: Asignaciones, comparaciones, operaciones aritméticas, acceso a arreglos.

Debemos dar su función de crecimiento...

SelectionSort

Veamos:

$$T(n) = c_1 + \text{nro. de opns. básicas del loop externo}$$

$$T(n) = c_1 + \sum_{i=0}^{n-1}(c_2 + \text{nro. de opns. básicas del loop interno})$$

$$T(n) = c_1 + \sum_{i=0}^{n-1}\left(c_2 + \sum_{j=i}^{n-1} c_3\right)$$

$$T(n) = c_1 + \sum_{i=0}^{n-1}(c_2 + (c_3(n-i)))$$

$$T(n) = c_1 + (nc_2) + c_3 \sum_{i=0}^{n-1}(n-i)$$

$$T(n) = c_1 + (nc_2) + c_3 n^2 - \left(\frac{n^2-n}{2}\right)$$

En este caso se dice que el algoritmo es orden cuadrático.

Recordar:

$$\sum_{i=k}^{n} c = ((n-k)+1) * c$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \quad c \neq 1$$

$$\sum_{i=k}^{n} (t+t') = \sum_{i=k}^{n} t + \sum_{i=k}^{n} t'$$

$$\sum_{i=k}^{n} c * t = c * \sum_{i=k}^{n} t$$

Orden de Crecimiento

Podemos clasificar los programas según su tasa de crecimiento

- $O(f(n))$: funciones que están acotadas por arriba por f
- $\Omega(f(n))$: funciones que están acotadas por abajo por f
- $\Theta(f(n))$: funciones que crecen exactamente como f

Notación O

$O(f(n))$ es la colección de funciones con una tasa de crecimiento menor o igual f

$t(n) \in O(g(n))$ ssi existen una constante positiva c y un entero no negativo n_0 tales que $t(n) \leq cg(n)$, para todo $n \geq n_0$.

Ejemplos:

$$n \in O(n^2)$$
$$100n + 5 \in O(n^2)$$
$$\frac{1}{2}n(n-1) \in O(n^2)$$
$$0.000001n^3 \notin O(n^2)$$

Notación Ω

$\Omega(g(n))$ es la colección de funciones con una tasa de crecimiento mayor o igual g.

$t(n) \in \Omega(g(n))$ ssi existen una constante positiva c y un entero no negativo n_0 tales que $t(n) \geq cg(n)$, para todo $n \geq n_0$

Ejemplos:

$$0.000001n^3 \in \Omega(n^3)$$
$$100n + 5 \notin \Omega(n^2)$$
$$\frac{1}{2}n(n-1) \in \Omega(n^2)$$

Notación Θ

$\Theta(f(n))$ Es la clase de funciones con un crecimiento exactamente igual a f

$t(n) \in \Theta(g(n))$ ssi existen constantes positivas c_1, c_2 y un entero no negativo n_0 tales que $c_1 g(n) \leq t(n) \leq c_2 g(n)$, para todo $n \geq n_0$

Ejemplos:

$$n \notin \Theta(n^2)$$
$$100n + 5 \notin \Theta(n^2)$$
$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$
$$0.000001n^3 \notin \Theta(n^2)$$

Algunas Clases Importantes

Las notaciones O Θ Ω permiten clasificar los programas según su orden, hay infinitas clases de funciones de crecimiento, pero algunas importantes:

- 1 :Algoritmos <u>constantes</u>, no dependen de la entrada
- n :Algoritmos <u>lineales</u>, recorren la entrada una cantidad constante de veces.
- $n * log\ n$:Varios algoritmos de sorting caen en esta clase

Algunas Clases Importantes (cont)

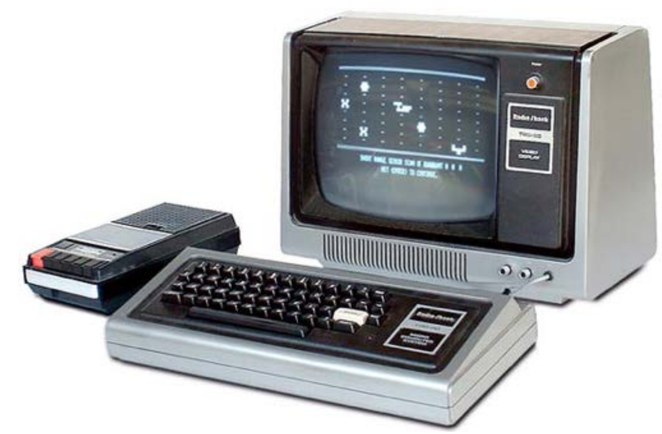
- n^2:Algoritmos <u>cuadráticos</u>, por cada elemento recorren una vez la entrada
- n^3:Algoritmos <u>cúbicos</u>, tres ciclos anidados.
- 2^n:<u>Exponencial</u> resuelven un problema utilizando varias instancias menores del mismo
- $n!$:<u>Factorial</u> tratan todas las combinaciones posibles

Comparación

ent.	$log_2 n$	n	$n log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20	10^6	2×10^7	10^{12}	10^{18}		

La Tasa de Crecimiento es más importante...

vs



ent.	Cray-1 Fortran Algoritmo Cubico	TRS-80 Basic Algoritmo Lineal
10	3 microseg.	200 miliseg.
100	3 miliseg.	2 seg.
1000	3 seg.	20 seg.
10000	50 seg.	50 seg.
100000	49 min.	3.2 min.
1000000	95 años	5.4 horas

Análisis en el Peor Caso

- Las operaciones básicas toman tiempo constante.
- El tiempo de $S_1; S_2$ es: $T_{S_1} + T_{S_2}$
- El tiempo de $If\ \ B\ \ then\ S_1\ \ else\ S_2$ es: $Max\{T_{S_1}, T_{S_2}\}$

Hay que considerar el tiempo de la condición

Análisis de Ciclos

El tiempo que tarda un ciclo es el tiempo que tarda el código de adentro multiplicado por la cantidad de iteraciones.

```
for (int i=0; i < n; i++) {
    for (int j=0; i < m, j++){
        x=x*x;
    }
}
```

Viene dado por:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} c = n * m * c$$

Ejemplo

```
public static boolean containsDuplicates(String[] args) {
    for (int i = 0; i < args.length; i++) {
        for (int j = 0; j < args.length; j++) {
            if (args[i] == args[j] && i!=j)
                return true;
        }
    }
return false;
}
```

Dice si un arreglo contiene duplicados o no

El tiempo viene dado por:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} c = n * n * c = n^2 * c$$

Es decir, el algoritmo es: $\Theta(n^2)$

Propiedades

Propiedades de O:

- Reflexividad: $f(n) \in O(f(n))$
- Transitividad: $f(n) \in O(g(n))$ y $g(n) \in O(t(n)) \Rightarrow f(n) \in O(t(n))$
- Sumas: Si $f(n) \in O(g(n) + t(n))$, entonces $f(n) \in O(Max(g(n), t(n)))$
- Mult. Constantes: Si $f(n) \in O(c * g(n))$, entonces $f(n) \in O(g(n))$
- Constantes: $k \in O(1)$ para cualquier constante k

Más Propiedades

Podemos usar límites para analizar funciones:

$$\text{Si } \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \text{ con } 0 < c, \text{ entonces } f(n) \in \theta(g(n))$$

$$\text{Si } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, \text{ entonces } f(n) \in O(g(n)) \text{ y } f(n) \notin \theta(g(n))$$

$$\text{Si } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty, \text{ entonces } f(n) \in \Omega(g(n)) \text{ y } f(n) \notin \theta(g(n))$$

Un Ejemplo

Demostremos: $log\ n \in O(n)$

$$\lim_{n\to\infty} \frac{log\ n}{n}$$

$$= [\text{Derivando ambos lados}]$$

$$\lim_{n\to\infty} \frac{\frac{log\ e}{n}}{1}$$

Recordar: $(log\ n)' = \dfrac{log\ e}{n}$

$$= [\text{Aritmetica}]$$

$$\lim_{n\to\infty} \frac{log\ e}{n} = 0$$

Algoritmos Recursivos

La técnica principal es expansión de recurrencias:

- La función de tiempo viene expresada por una ecuación recursiva.
- Para resolverlas tenemos que utilizar substituciones
- Los casos bases nos permiten terminar este proceso de substitución.

Un Ejemplo

Consideremos:

```
public void ordenar(int A[], int i, int j) {
    if (i<=j) {
        int ind = i; boolean ordenado = false;
        ordenar(A, i+1, j);
        while ((ind<j) & !ordenado) {
            if (A[ind]>A[ind+1]) {
                int aux = A[ind+1];
                A[ind+1] = A[ind];
                A[ind] = aux;
            }
            else {
                ordenado = true;
            }
            ind++;
        }
    }
}
```

Método de ordenamiento recursivo

Peor caso: ordenado decrecientemente

Ejemplo (cont)

Veamos las ecuaciones de recurrencia:

$$T(0) = 1$$

El while se repite n veces

$$T(n) = 1 + T(n-1) + \sum_{i=0}^{n-1} 1$$

La llamada recursiva

Simplificando:

$$T(n) = T(n-1) + n + 1$$

Expansión de Recurrencias

Podemos expandir las ecuaciones:

$$
\begin{aligned}
T(n) &= T(n-1) + n + 1 \\
T(n) &= [T(n-2) + (n-1) + 1] + n + 1 \\
T(n) &= T(n-2) + (n-1) + n + 2 \\
T(n) &= [T(n-3) + (n-2) + 1] + (n-1) + n + 2 \\
T(n) &= T(n-3) + (n-2) + (n-1) + n + 3 \\
&\vdots \\
T(n) &= T(n-i) + \textstyle\sum_{j=0}^{i-1}(n-j) + i
\end{aligned}
$$

Resolviendo las Ecuaciones

Podemos reemplazar $T(n-i)$ por 1 en la ecuación cuando

$$n - i = 0$$

es decir:

$$i = n$$

Es decir, en la ecuación obtenemos:

$$T(n) = 1 + \sum_{j=0}^{n-1}(n-j) + n = 1 + n^2 + \frac{(n^2 - n)}{2}$$

Usando las propiedades, el algoritmo es $O(n^2)$.En realidad también tenemos $\Theta(n^2)$

Otro Ejemplo

Consideremos devuelta el método:

```
/**
* Version ineficiente de f(n)=2^n
*/
public static long exp1(long n){
    if (n == 0)
        return 1;
    else
        return exp1(n-1) + exp1(n-1);
}
```

Hace dos llamadas recursivas por cada predecesor de n

$$T(0) = 1$$
$$T(n) = 2 * T(n-1) + c$$

Ecuación de recurrencia

Calculemos

$$
\begin{aligned}
T(n) &= 2 * T(n-1) + c \\
&= 2 * [2 * T(n-2) + c] + c \\
&= 2 * [2 * [2 * T(n-3) + c] + c] + c \\
&\vdots \\
&= 2^i * T(n-i) + \sum_{j=0}^{i-1} c * 2^j \\
&= 2^n * c + \sum_{j=0}^{n-1} c * 2^j \\
&= 2^n * c + c * (2^n - 1) \quad \in \Theta(2^n)
\end{aligned}
$$

Otra versión

Otra versión más eficiente del mismo algoritmo:

```
/**
 * Version mas eficiente de f(n) = 2^n
 */
 public static long exp2(long n){
     if (n == 0)
         return 1;
     else
         return 2*exp2(n-1);
 }
```

Hace solo una llamada recursiva por cada predecesor de n

$$T(0) = c$$
$$T(n) = c + T(n-1)$$

Ecuación de recurrencia

Resolvamos la Ecuación

$$
\begin{aligned}
T(n) &= c + T(n-1) \\
&= c + [c + T(n-2)] \\
&= c + [c + [c + T(n-3)]] \\
&\vdots \\
&= i * c + T(n-i) \\
&= n * c + c \ \in \Theta(n)
\end{aligned}
$$

Cuando n-i=0

Algoritmo lineal, mucho más eficiente!

Fibonacci

Veamos Fibonacci:

$$fib\,0 = 1$$
$$fib\,1 = 1$$
$$fib\,n = fib\,(n-1) + fib\,(n-2)$$

Su tiempo de ejecución viene dado por:

$$T(0) = 0$$
$$T(1) = 1$$
$$T(n) = T(n-1) + T(n-2) + c$$

Su tiempo de ejecución viene dado por Fibonacci!

Fibonacci (cont.)

El tiempo exacto de Fibonacci no es directo, acotemos por abajo y arriba

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\leq T(n-1) + T(n-1) + c \\
&= 2 * T(n-1) + c \\
&\quad \vdots \\
&= 2^i * T(n-i) + i * c \\
&= [\text{ con } i = n-1] \\
& 2^{n-1} + (n-1) * c
\end{aligned}
$$

Es decir: $T(n) \in O(2^n)$

Fibonacci

Si acotamos por abajo:

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\geq T(n-2) + T(n-2) + c \\
&= 2 * T(n-2) + c \\
&\vdots \\
&= 2^i * T(n - 2 * i) + i * c \\
&= [\text{ con } i = \frac{n}{2}] \\
&2^{n/2} + \frac{n}{2} * c
\end{aligned}
$$

Es decir: $T(n) \in \Omega(2^{n/2})$

Tiempo Exacto de Fibonacci

Consideremos los siguientes números:

$$\phi = \frac{1+\sqrt{5}}{2} \qquad \text{y} \qquad \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

Se puede demostrar que:

$$fib(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

Es decir: $fib(n) \in \theta(\phi^n)$ en donde $\phi \cong 1.61$